

Programación Orientada a Objetos (OOP)

Terminología

- Cada **objeto** creado en un programa es una **instancia** de una **clase**
- Cada clase presenta al mundo exterior una vista concisa y coherente de los objetos que son instancias de esta clase, sin entrar en demasiados detalles innecesarios, ni dar acceso a otros al funcionamiento interno de los objetos.
- La definición de clase normalmente especifica **las variables de instancia**, también conocidas como **miembros de datos**, que contiene el objeto, así como **los métodos**, también conocidos como **funciones miembro**, que el objeto puede ejecutar.

Principios y Metas

- Robustez
 - Queremos que el software sea capaz de manejar entradas inesperadas que no están explícitamente definidas para su aplicación.
- Adaptabilidad
 - El software debe ser capaz de evolucionar con el tiempo en respuesta a las condiciones cambiantes de su entorno.
- Reutilización
 - El mismo código debe poder utilizarse como componente de diferentes sistemas en varios aplicaciones

Tipos Abstractos de Datos

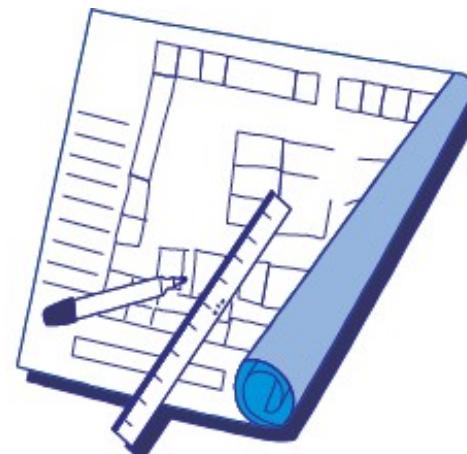
- **La abstracción** es sintetizar un sistema hasta sus partes más fundamentales.
- La aplicación del paradigma de abstracción al diseño de estructuras de datos da lugar a los **tipos abstractos de datos** (TDA's).
- Un TDA es un modelo de una estructura de datos que especifica el **tipo** de datos almacenados, las **operaciones** admitidas en ellos y los tipos de parámetros de las operaciones.
- Un TDA especifica lo que hace cada operación, pero no cómo lo hace.
- El conjunto colectivo de comportamientos soportados por un TDA es su **interfaz pública**

Principios del Diseño Orientado a Objetos

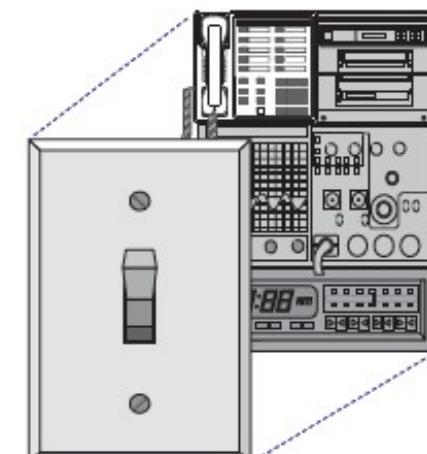
- Modularidad
- Abstracción
- Encapsulación



Modularity

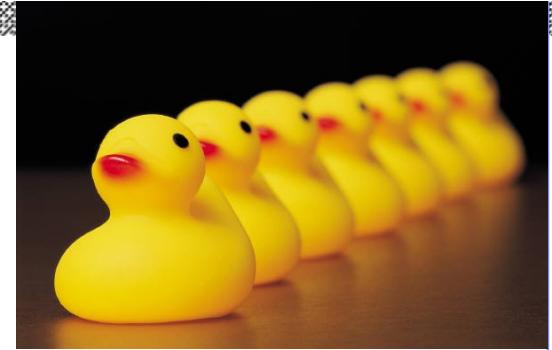


Abstraction



Encapsulation

Duck Typing



- Python trata las abstracciones implícitamente mediante un mecanismo conocido como **duck typing** ("tipado a lo pato")
 - Un programa puede tratar los objetos como si tuvieran cierta funcionalidad y se comportarán correctamente siempre que esos objetos proporcionen la funcionalidad esperada.
- Como un lenguaje interpretado y tipificado dinámicamente, no hay verificación en "tiempo de compilación" de los tipos de datos en Python, y no hay un requisito formal para las declaraciones de clase de base abstractas.
- El término "*duck typing*" proviene de un adagio atribuido al poeta James Whitcomb Riley, que afirma que "*cuando veo un pájaro que camina como un pato, nadá como un pato y grazna como un pato, llamo a ese pájaro pato.*"

Clases Abstractas de Base



- Python admite tipos de datos abstractos mediante un mecanismo conocido como **clase abstracta base (CAB)**.
- No se puede crear una instancia de una clase abstracta base, pero define uno o más métodos comunes que todas las implementaciones de la abstracción deben tener.
- Una CAB se realiza mediante una o más clases concretas que heredan de la clase abstracta base y proporcionan implementaciones para los métodos declarados en la CAB.
- Podemos hacer uso de varias clases abstractas existentes provenientes de la colección de módulos de Python, que incluye definiciones a varios TAD's para estructuras de datos comunes e implementaciones concretas de algunas de estas.

Encapsulación

- Otro principio importante del diseño orientado a objetos es la **encapsulación**.
 - Los diferentes componentes de un sistema de software no deben revelar los detalles internos de sus respectivas implementaciones.
- Algunos aspectos de una estructura de datos son públicos y otros están destinados a ser detalles internos.
- Python solo proporciona escaso soporte para la encapsulación
 - Por convención, se supone que los nombres de los miembros de una clase (tanto los datos como funciones) que comienzan con un guión bajo (por ejemplo, `_secret`, `_privado`) no son públicos y no se debe confiar en ellos.



Patrones de Diseño

- **Diseño de Algoritmos:**
 - Recursividad
 - Amortización
 - Divide y Conquistaras
 - Podar y Buscar
 - Fuerza Bruta
 - Programación Dinámica
 - El Método “*Greedy*”
- **Diseño de Software:**
 - Iterador
 - Adaptador / Wrapper
 - Posición
 - Composición
 - Métodos Plantilla
 - Factory method

Diseño de Software Orientado a Objetos

- **Responsabilidades** : Divida el trabajo en diferentes actores, cada uno con una responsabilidad diferente.
- **Independencia** : defina el trabajo de cada clase para que sea lo más independiente posible de otras clases .
- **Comportamientos** : defina los comportamientos de cada clase con cuidado y precisión, de modo que las consecuencias de cada acción realizada por una clase sean bien entendidas por otras clases que interactúan con él.

Lenguaje de Modelado Unificado (UML)

Un **diagrama de clases** tiene tres porciones

1. El **nombre** de la clase.
2. Las **variables de instancia** recomendadas.
3. Los **métodos** recomendados de la clase.

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer() get_bank() get_account() make_payment(amount)	get_balance() get_limit() charge(price)

Definiciones de Clase

- Una **clase** sirve como medio principal para la abstracción en la orientación a objetos. programación.
- En Python, cada dato se representa como una **instanciа** de algún clase.
- Una clase proporciona un conjunto de comportamientos en forma de funciones (también conocidas como **métodos**), con implementaciones que pertenecen a todas sus instancias.
- Una clase también sirve como modelo para sus instancias, determinando efectivamente la forma en que la información de estado para cada instancia se representa en forma de **atributos** (también conocidos como **campos** , **variables de instancia** o simplemente **datos**).

El identificador ***self***

- En Python, el "autoidentificador" ***self*** juega un rol fundamental identificando al objeto.
- Para cualquier *clase*, posiblemente puede haber muchas instancias diferentes, y cada una debe mantener sus propias variables de instancia.
- Por lo tanto, cada instancia almacena sus propias variables de instancia para reflejar su estado actual. Sintácticamente, ***self*** identifica la instancia sobre la cual se aplica un método invocado.

Ejemplo

```
1 class CreditCard:  
2     """A consumer credit card."""  
3  
4     def __init__(self, customer, bank, acnt, limit):  
5         """Create a new credit card instance.  
6  
7         The initial balance is zero.  
8  
9         customer  the name of the customer (e.g., 'John Bowman')  
10        bank      the name of the bank (e.g., 'California Savings')  
11        acnt      the account identifier (e.g., '5391 0375 9387 5309')  
12        limit      credit limit (measured in dollars)  
13        """  
14        self._customer = customer  
15        self._bank = bank  
16        self._account = acnt  
17        self._limit = limit  
18        self._balance = 0  
19
```

Ejemplo, Parte 2

```
20  def get_customer(self):
21      """Return name of the customer."""
22      return self._customer
23
24  def get_bank(self):
25      """Return the bank's name."""
26      return self._bank
27
28  def get_account(self):
29      """Return the card identifying number (typically stored as a string)."""
30      return self._account
31
32  def get_limit(self):
33      """Return current credit limit."""
34      return self._limit
35
36  def get_balance(self):
37      """Return current balance."""
38      return self._balance
```

Ejemplo, Parte 3

```
39  def charge(self, price):
40      """Charge given price to the card, assuming sufficient credit limit.
41
42      Return True if charge was processed; False if charge was denied.
43      """
44      if price + self._balance > self._limit:      # if charge would exceed limit,
45          return False                            # cannot accept charge
46      else:
47          self._balance += price
48          return True
49
50  def make_payment(self, amount):
51      """Process customer payment that reduces balance."""
52      self._balance -= amount
```

Constructores

- Un usuario puede crear una instancia de la clase *CreditCard* usando:

```
cc = CreditCard('John Doe', '1st Bank', '5391 0375 9387 5309', 1000)
```
- Internamente, esto da como resultado una llamada al método de __init__ con nombre que sirve como el **constructor** del la clase.
- Su principal responsabilidad es establecer el estado de un objeto recién creado con variables de instancia

Sobrecarga de Operadores

- Las Clases integradas de Python proporcionan la semántica natural para muchos operadores.
- Por ejemplo, la sintaxis $\mathbf{a} + \mathbf{b}$ invoca la *suma* para tipos numéricos, pero la *concatenación* para tipos contenedores.
- Al definir una nueva clase, debemos considerar si se debe definir una semántica (operacional) diferente para $\mathbf{a} + \mathbf{b}$ cuando \mathbf{a} o \mathbf{b} es una instancia de esa clase.

Sobrecarga de Operadores (Cont.)

Common Syntax	Special Method Form
$a + b$	<code>a.__add__(b);</code> alternatively <code>b.__radd__(a)</code>
$a - b$	<code>a.__sub__(b);</code> alternatively <code>b.__rsub__(a)</code>
$a * b$	<code>a.__mul__(b);</code> alternatively <code>b.__rmul__(a)</code>
a / b	<code>a.__truediv__(b);</code> alternatively <code>b.__rtruediv__(a)</code>
$a // b$	<code>a.__floordiv__(b);</code> alternatively <code>b.__rfloordiv__(a)</code>
$a \% b$	<code>a.__mod__(b);</code> alternatively <code>b.__rmod__(a)</code>
$a ** b$	<code>a.__pow__(b);</code> alternatively <code>b.__rpow__(a)</code>
$a << b$	<code>a.__lshift__(b);</code> alternatively <code>b.__rlshift__(a)</code>
$a >> b$	<code>a.__rshift__(b);</code> alternatively <code>b.__rrshift__(a)</code>
$a \& b$	<code>a.__and__(b);</code> alternatively <code>b.__rand__(a)</code>
$a ^ b$	<code>a.__xor__(b);</code> alternatively <code>b.__rxor__(a)</code>
$a b$	<code>a.__or__(b);</code> alternatively <code>b.__ror__(a)</code>
$a += b$	<code>a.__iadd__(b)</code>
$a -= b$	<code>a.__isub__(b)</code>
$a *= b$	<code>a.__imul__(b)</code>
...	...
$+a$	<code>a.__pos__()</code>
$-a$	<code>a.__neg__()</code>
$\sim a$	<code>a.__invert__()</code>
$abs(a)$	<code>a.__abs__()</code>
$a < b$	<code>a.__lt__(b)</code>
$a <= b$	<code>a.__le__(b)</code>
$a > b$	<code>a.__gt__(b)</code>
$a >= b$	<code>a.__ge__(b)</code>
$a == b$	<code>a.__eq__(b)</code>
$a != b$	<code>a.__ne__(b)</code>
$v \text{ in } a$	<code>a.__contains__(v)</code>
$a[k]$	<code>a.__getitem__(k)</code>
$a[k] = v$	<code>a.__setitem__(k,v)</code>
$del a[k]$	<code>a.__delitem__(k)</code>
$a(arg1, arg2, ...)$	<code>a.__call__(arg1, arg2, ...)</code>
$len(a)$	<code>a.__len__()</code>

Iteradores

- La Iteración es un concepto importante en el diseño de datos. estructuras
- Un *iterador* para una colección proporciona comportamiento clave:
 - Es compatible con un método especial llamado **next** que devuelve el siguiente elemento de la colección, si la hay, o genera una excepción *StopIteration* para indicar que no hay más elementos.

Iteradores Automáticos

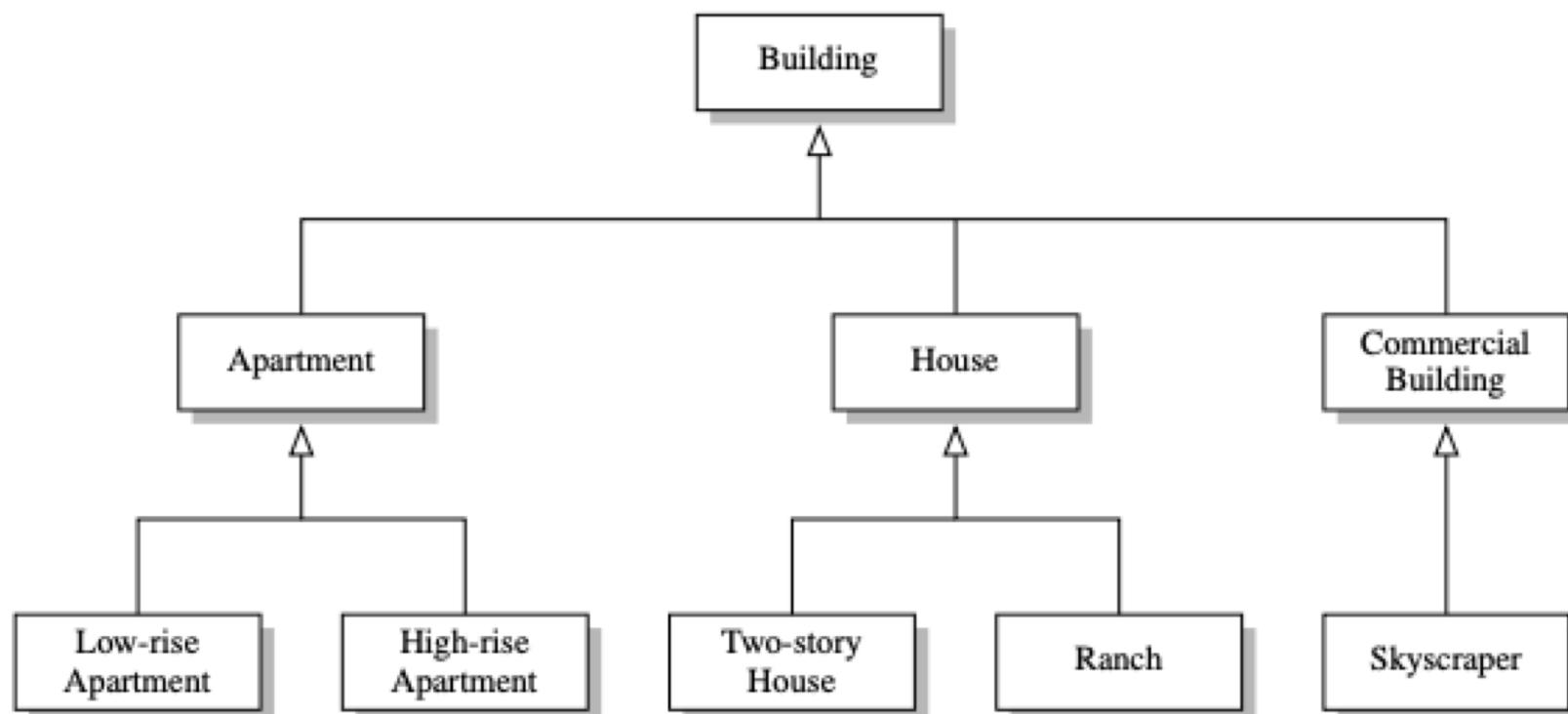
- Python también ayuda proporcionando una implementación de iterador automático para cualquier clase que defina los métodos **len** y **getitem**

```
1 class Range:  
2     """A class that mimics the built-in range class."""  
3  
4     def __init__(self, start, stop=None, step=1):  
5         """Initialize a Range instance.  
6  
7         Semantics is similar to built-in range class.  
8         """  
9         if step == 0:  
10             raise ValueError('step cannot be 0')  
11  
12         if stop is None:                      # special case of range(n)  
13             start, stop = 0, start            # should be treated as if range(0,n)  
14  
15         # calculate the effective length once  
16         self._length = max(0, (stop - start + step - 1) // step)  
17  
18         # need knowledge of start and step (but not stop) to support __getitem__  
19         self._start = start  
20         self._step = step  
21  
22     def __len__(self):  
23         """Return number of entries in the range."""  
24         return self._length  
25  
26     def __getitem__(self, k):  
27         """Return entry at index k (using standard interpretation if negative)."""  
28         if k < 0:  
29             k += len(self)                  # attempt to convert negative index  
30  
31         if not 0 <= k < self._length:  
32             raise IndexError('index out of range')  
33  
34         return self._start + k * self._step
```

Herencia

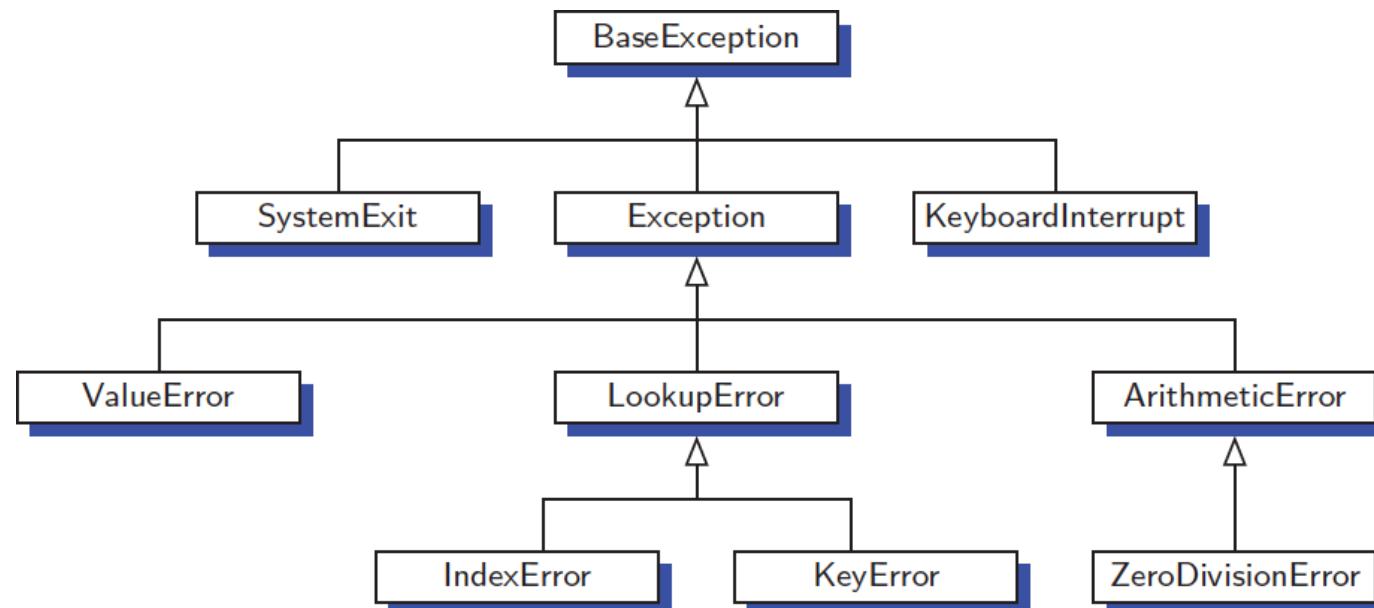
- Un mecanismo para una organización modular y jerárquica es la **herencia**.
- Esto permite definir una nueva clase basada en una clase existente como punto de partida.
- La clase existente generalmente se describe como la **clase base**, *clase principal*, o la *superclase*, mientras que la clase recién definida se conoce como la **subclase** o clase hijo/a.
- Hay dos formas en que una subclase puede diferenciarse de su superclase:
 - Una subclase puede **especializar** un comportamiento existente al proporcionar una nueva implementación que sobrecarga el comportamiento del método existente.
 - Una subclase también puede **extender** su superclase proporcionando métodos completamente nuevos.

Herencia (Cont.)



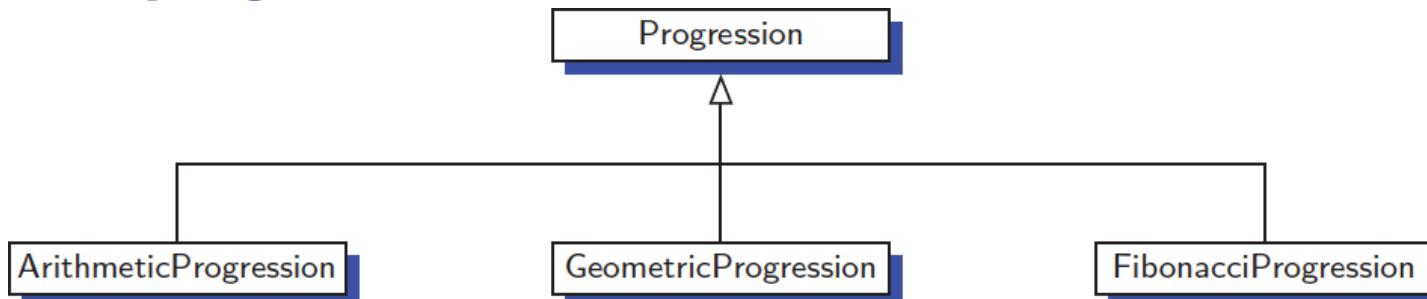
La herencia está incorporada en Python

- Una parte de la jerarquía de excepción de Python tipos:



Ejemplo extendido

- Una **progresión numérica** es una secuencia de números, donde cada número depende de uno o más de los anteriores . números.
 - Una **progresión aritmética** determina el siguiente número sumando una constante fija al anterior valor.
 - Una **progresión geométrica** determina el siguiente número multiplicando el valor anterior por un número fijo . constante.
 - Una **progresión de Fibonacci** utiliza la fórmula $N_{i+1} = N_i + N_{i-1}$



La Clase base de progresión

```
1 class Progression:
2     """Iterator producing a generic progression.
3
4     Default iterator produces the whole numbers 0, 1, 2, ...
5     """
6
7     def __init__(self, start=0):
8         """Initialize current to the first value of the progression."""
9         self._current = start
10
11    def _advance(self):
12        """Update self._current to a new value.
13
14        This should be overridden by a subclass to customize progression.
15
16        By convention, if current is set to None, this designates the
17        end of a finite progression.
18        """
19        self._current += 1
20
21    def __next__(self):
22        """Return the next element, or else raise StopIteration error."""
23        if self._current is None:      # our convention to end a progression
24            raise StopIteration()
25        else:
26            answer = self._current      # record current value to return
27            self._advance()           # advance to prepare for next time
28            return answer             # return the answer
29
30    def __iter__(self):
31        """By convention, an iterator must return itself as an iterator."""
32        return self
33
34    def print_progression(self, n):
35        """Print next n values of the progression."""
36        print(' '.join(str(next(self)) for j in range(n)))
```

Progresión aritmética subclase

```
1 class ArithmeticProgression(Progression):      # inherit from Progression
2     """ Iterator producing an arithmetic progression."""
3
4     def __init__(self, increment=1, start=0):
5         """Create a new arithmetic progression.
6
7             increment  the fixed constant to add to each term (default 1)
8             start      the first term of the progression (default 0)
9
10        """
11        super().__init__(start)                  # initialize base class
12        self._increment = increment
13
14    def _advance(self):                      # override inherited version
15        """Update current value by adding the fixed increment."""
16        self._current += self._increment
```

Progresión geométrica subclase

```
1 class GeometricProgression(Progression):      # inherit from Progression
2     """ Iterator producing a geometric progression."""
3
4     def __init__(self, base=2, start=1):
5         """Create a new geometric progression.
6
7             base      the fixed constant multiplied to each term (default 2)
8             start    the first term of the progression (default 1)
9         """
10    super().__init__(start)
11    self._base = base
12
13    def _advance(self):                      # override inherited version
14        """Update current value by multiplying it by the base value."""
15        self._current *= self._base
```

FibonacciProgresión subclase

```
1 class FibonacciProgression(Progression):
2     """Iterator producing a generalized Fibonacci progression."""
3
4     def __init__(self, first=0, second=1):
5         """Create a new fibonacci progression.
6
7             first      the first term of the progression (default 0)
8             second    the second term of the progression (default 1)
9
10            """
11            super().__init__(first)                  # start progression at first
12            self._prev = second - first            # fictitious value preceding the first
13
14    def _advance(self):
15        """Update current value by taking sum of previous two."""
16        self._prev, self._current = self._current, self._prev + self._current
```